Fostering SLA-Driven API Specifications

Antonio Gamez-Diaz[1], Pablo Fernandez[1], and Antonio Ruiz-Cortes[1]

Universidad de Sevilla*,
{agamez2, pablofm, aruiz}@us.es

Abstract. Software architecture tendencies are shifting to a microservice paradigm. In this context, RESTful APIs are being established the standard of integration. API designer often identifies two key issues to be competitive in such growing market. On the one hand, the generation of accurate documentation of the behavior and capabilities of the API to promote its usage; on the other hand, the design of a pricing plan that fits into the potential API user's needs.
Besides the increasing number of API modeling alternatives is emerging, there is a lack of proposals on the definition of flexible pricing plans usually contained in the Service Level Agreements (SLAs).
In this paper we propose two different modeling techniques for the description of SLA in a RESTful API context: iAgree and SLA4OAI.

1 Introduction

In recent years, there has been a trend towards a new architectural style that has been called microservice. This style requires that each component (a microservice) can evolve, scale and deploy independently to the rest, increasing the flexibility of the system as a whole; in fact, this approach has been the architectural style of choice in demanding web applications such as eBay, Amazon and Netflix [7].

From an engineering perspective, a key element of these architectures with respect to the modeling and implementation of microservices is the usage of the RESTful paradigm. This paradigm provides a unified approach to identify the granularity and operational interface of microservices that has a high degree of extensibility. In particular, RESTful provides a lighter approach for building, deploying and scaling microservices more effectively. Moreover, this approach is aligned with the fundamental web principles as part of the HyperText Transfer Protocol (HTTP). This microservice tendency represents a recent shift in software engineering towards API-driven development; in such context, a fast feedback on the API design can be facilitated by modeling and visualizing the designed API behavior during early stages of the development process.

* This work has been partially supported by the European Commission (FEDER), the Spanish and the Andalusian R&D&I programs (grants TIN2015-70560-R (BELI), P12–TIC-1867 (COPAS) and and TIN2014-53986-REDT (RCIS)) the FPU scholarship program, granted by the Spanish Ministry of Education, Culture and Sports (FPU15/02980).

As APIs are gaining notoriety and API marketplaces are increasingly growing, at least two key aspects can be identified: i) *ease of use* for its potential developers; ii) a flexible usage *plan* that fits their customer's demands.

Regarding the *ease of use* perspective, whereas the awareness for the need of documenting the low-level HTTP details of the static structure of RESTful APIs has resulted with several tools, such as Swagger/OpenAPI, there is still a lack of support for conceptual modeling and visualization of REST API's dynamics. Furthermore, reaching a certain resource state frequently requires undertaking a predefined sequence of interactions or choosing among different alternative paths, thus shifting from the concept of a single RESTful interaction to the concept of a RESTful conversation. As conversations become more complex, visualizing them (e.g., by means of modeling techniques such as BPMN choreographies or RESTalk) can help decrease the cognitive load for both API designers and clients, who need to communicate their designs and understand how to correctly use the API.

Conversely, from the *plans* perspective, to the best of our knowledge, there does not exist a widely accepted model to describe usage plans including elements such as cost, functionality restrictions or invocation rate limits. In this context, authors have proposed the SLA4OAI extension in the OAI Community in order to address this issues. On the other hand, a number of API management platforms, commonly known as *API Gateways* [3], have tried to address the problem of modeling usage plans but they are typically constrained by their platform architecture and do not provide an inter-operable nor vendor-neutral usage plan specification. A formal definition of these plans can foster new enticing challenges, such as the automatic analysis of optimal cloud offerings [4] or analysis of temporal aspects [6].

Despite the fact that there exists some modeling techniques for describing API documentation (e.g. OpenAPI specification), pricing plans for APIs (e.g. SLA4OAI) and RESTful conversations (e.g. RESTalk), to the best of our knowledge, there is not a unified model able to integrate all of these modeling viewpoints.

The rest of this paper is structured as follows: Section 2 describes the iAgree proposal for modeling general SLAs in different ecosystems. Conversely, Section 3 presents SLA4OAI, a proposal for describing SLAs in the context of RESTful APIs. Next, Section 4 illustrates the SLA modeling with iAgree and SLA4OAI through an example. Finally, Section 5 summarizes and identifies the challenges to be addressed.

2 iAgree

iAgree[1] is an open source language for describing Service Level Agreements (SLAs) in a vendor-neutral way. iAgree is intended to model, including, but not limited to, two main scenarios: computational services (e.g., RESTful APIs) and

[1] http://iagree.specs.governify.io/

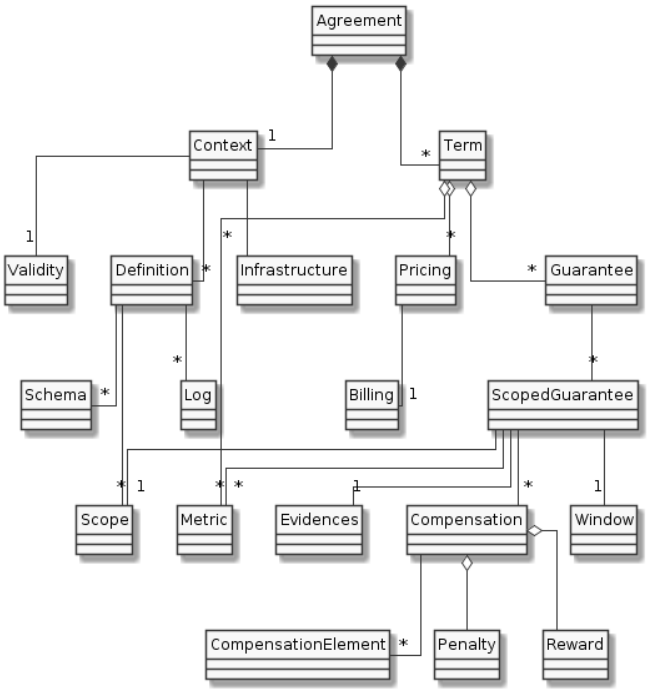

Fig. 1. iAgree elements.

human services (e.g., business processes). iAgree has evolved notoriety since the preliminary versions described in [8].

An iAgree description is a JSON or a YAML document based upon JSON-Schema[2] with the structure defined in Figure 1.

Specifically, an iAgree model typically includes the following parts:

SLA Object: every iAgree document contains the following sections: id (unique identification), version (document version), type (type based on the SLA life-cycle), context (holds the main information of the SLA context) and terms (holds the main information of the SLA terms).

ContextObject: holds the main information of the SLA context. iAgree (iAgree specification version), provider (owner/host of the service), consumer (entity that consumes the service), infrastructure (provides information about the tools used for SLA storage, calculation, governance,), validity (holds the main information of the SLA validity) and definitions (holds the main information of the SLA definitions).

ValidityObject: holds the main information of the SLA validity. initial (start date of the SLA according to ISO 8601 time format), timeZone (time zone

[2] http://json-schema.org

of the SLA according to ISO 8601 time zone format) and end (end date of the SLA according to ISO 8601 time format).

DefinitionsObject: holds the main information of the SLA definitions. schemas (definition schemas) and scopes (definition scopes).

TermsObject: holds the main information of the SLA terms. pricing (holds the main information of the SLA pricing), metrics (holds the main information of the SLA metric.), guarantees (holds the main information of the SLA guarantees.), configurations (holds the main information of the SLA configurations.), quotas (holds the main information of the SLA quotas.), rates (holds the main information of the SLA rates).

PricingObject: holds the main information of the SLA pricing. cost (cost associated to this service), currency (currency used to express the cost according to ISO 4217. Samples: USD, EUR, or BTC for US dollar, euro, or bitcoin, respectively) and billing (holds the main information of the SLA billing).

BillingObject: holds the main information of the SLA billing. period (period used for billing. Supported values are: onepay: unique payment before start using the service; daily: billing at the end of every day; weekly: billing at the end of every week; monthly: billing at the end of every month; quarterly: billing at the end of every quarter; yearly: billing at the end of every year), initial (start date of the billing cycle according to ISO 8601 time format), penalties (holds the main information of the SLA billing penalties) and rewards (holds the main information of the SLA billing rewards).

MetricsObject: holds the main information of the SLA metrics. metridId (holds the main information of an SLA single metric).

MetricObject: holds the main information of the SLA single metric. schema, type and scope

CompensationObject: holds the main information of the SLA single compensation. over (metrics involved in the compensation calculation process), of (holds the main information of the SLA scoped compensations), aggegatedBy (compensation aggregation function), groupBy (compensation aggrupation function), upTo (compensation limit)

ScopedCompensationObject: holds the main information of the SLA single scoped compensation. value (scoped compensation value) and condition (scoped compensation condition).

GuaranteeObject: holds the main information of the SLA single scoped compensation. id (guarantee unique identification), scope (guarantee scope), of (holds the main information of the SLA scoped guarantees).

ScopedGuaranteObject: holds the main information of the SLA single scoped guarantee. scope (scoped guarantee scope), objective (guarantee objective), with (definition of metrics referenced in scope attribute), window (guarantee window), evidences (guarantee evidences), penalties (holds the main information of the SLA guarantee penalties), rewards (holds the main information of the SLA guarantee rewards).

WindowObject: holds the main information of the SLA guarantee window. initial (start date of the window according to ISO 8601 time), end (end date of the window according to ISO 8601 time), type (window type), period

(used period. Supported values are: daily: at the end of every day; weekly: at the end of every week; monthly: at the end of every month; quarterly: at the end of every quarter; yearly: at the end of every year).

ConfigurationsObject: holds the main information of the SLA configurations. configurationId (holds the main information of the SLA configurations).

ConfigurationObject: holds the main information of the SLA configurations. scope (configuration scope), of (holds the main information of the SLA scoped configuration).

ScopedConfigurationObject: holds the main information of the SLA single scoped configuration. scope (configuration scope), value (configuration value).

QuotaObject: holds the main information of the SLA single quota. id (quota unique identification), scope (quota scope), over (metrics involved in the quota calculation process), of (holds the main information of the SLA scoped quotas).

ScopedQuotaObject: holds the main information of the SLA single scoped quota. scope (scoped quota scope), limits (holds the main information of an SLA scoped quota limit).

LimitObject: holds the main information of the SLA scoped quota/rate limit. max (quota/rate maximum value), period (used period. Supported values are: daily: at the end of every day; weekly: at the end of every week; monthly: at the end of every month; quarterly: at the end of every quarter; yearly: at the end of every year).

RateObject: holds the main information of the SLA single rate. id (rate unique identification), scope (rate scope), over (metrics involved in the rate calculation process), of (holds the main information of the SLA scoped rates).

ScopedRateObject: holds the main information of the SLA single scoped rate. scope (scoped rate scope), limits (holds the main information of an SLA scoped rate limit).

3 SLA4OAI

A number of proposals have emerged towards formalizing API definitions, but they are rarely used in practice. Web Application Description Language [5], a specification language for RESTful APIs was the first one to be proposed. Other proposals such as API Blueprint[3], RAML[4], IODocs[5], or Swagger[6] also addressed this API formalization goal. However, they were not widely adopted in the industry [1].

Nonetheless, since the Swagger specification was released to the community major improvements have been made over the initial proposal, such as defining

[3] https://apiblueprint.org/
[4] https://raml.org
[5] https://github.com/mashery/iodocs
[6] https://swagger.io

links between resources or supporting different mechanisms of API invocation. This fact yielded the creation of the OpenAPI Initiative (OAI)[7], an open source consortium hosted by The Linux Foundation and supported by a growing number of leading industrial stakeholders (e.g., Google, IBM, SmartBear, PayPal or 3Scale). The OAI, therefore, is a vendor-neutral, portable and open specification for the creation, evolution, and promotion of a description format for APIs represented in JSON or YAML files.

An OpenAPI specification (OAS) document, typically contains several elements, namely: *API general information* (e.g., title, description and version); *API endpoints*, specifically the deployment server URL enabling, thereby, having multiple endpoints differencing production and developing stages by means of the combination of the deployment URL with the relative resource ; a number of *schemas*, defining common data structures used in the API.

Notwithstanding, the key concept in an OAS document is the `paths` section, in which it is defined each individual endpoint (i.e., resource path) and the HTTP methods (i.e., resource operations) supported by these endpoints. Operations can have parameters passed via URL path, query string, header or cookies. If an operation sends a request body, the body content and media type can be described as well.

An enticing feature of the OpenAPI Specification (OAS) is the capability of being extended with the definition of custom properties starting with `x-`. This fact paves the way for customizing or adding additional features according to specific business needs. Specifically, SLA4OAI represents an example of such extension capabilities. As depicted in Code 1.2, SLA4OAI[8] provides a model for describing SLA in APIs in a neutral vendor flavor way. This extension is intended support for different pricing aspects so that API management tools can import and measure key metrics and build SLAs for APIs in a standard way. This point is especially interesting in the context of industrial API Gateways since previous works point out that most real API providers apply limitations (such as quotas and rates limitations depending on the pricing plan subscribed by clients) [2].

We define an SLA-driven OAS as an OAS document which has been extended with an optional attribute, `x-sla`, with a URI pointing to the JSON or a YAML document that contains different sections, as depicted in the fragment shown in Code 1.2. Next, each section is briefly described beneath:

SLA Object: every SLA4OAI document contains the following sections: context (holds the main information of the SLA context), infrastructure (provides information about tooling used for SLA storage, calculation, governance, etc.), pricing (global pricing data), metrics (a list of metrics to use in the context of the SLA), plans (a set of plans to define different service levels per plan), quotas (global quotas, these are the default quotas, but they could be overridden by each plan late), rates (global rates, these are the default rates, but they could be overridden by each plan late), guarantees (global

[7] https://www.openapis.org
[8] https://github.com/isa-group/SLA4OAI-Specification

guarantees, these are the default guarantees, but they could be overridden by each plan late),configuration (define the default configurations, later each plan can override it).

ContextObject: id (the identification of the SLA context), version (indicates the version of the SLA format), api (indicates a URI (absolute or relative) describing the API to instrument described in the OpenAPI format), type (the type of SLA based on the life-cycle of agreement, that is, plans or instance), provider (provider information: data about the owner/host of the API. This field is required in case of the context type is instance), consumer (consumer information, data about the entity that consumes the service. This field is required in case of the context type is instance), validity (availability of the service expressed via time slots. This field is required in case of the context type is instance).

ValidityObject: effectiveDate (the starting date of the SLA agreement using the ISO 8601 time intervals format), expirationDate (the expiration date of the SLA agreement using the ISO 8601 time intervals format).

InfrastructureObject: supervisor (location of the SLA Check service endpoint), monitor (location of the SLA Metrics endpoint), name (optional endpoints of SLA governance infrastructure).

PricingObject: cost (cost associated to this service. Defaults to 0 if unspecified), currency (currency used to express the cost. Supported currency values are expressed in ISO 4217 format. Samples: USD, EUR, or BTC for US dollar, euro, or bitcoin, respectively. Defaults to USD if unspecified), billing (period used for billing. Supported values are: onepay Unique payment before start using the service. - daily Billing at end of the day. - weekly Billing at end of the week. - monthly Billing at end of the month. - quarterly Billing at end of the quarter. - yearly Billing at end of the year. Default to monthly if unspecified).

MetricsObject: name (definitions of metrics with name, types and descriptions), name (reference to pre-existing metrics in external file).

MetricObject: type (this is the metric type accordingly to the OAI specification format column), format (the extending format for the previously mentioned type. See Data Type Formats for further detail), description (a brief description of the metric), unit (the unit of the metric), resolution (determine when this metric will be resolved. If value checks the metric will be sent before the calculation of SLA state, else if value is consumption the metric will be sent after consumption).

PlansObject: planName (describes a usage plan for the API with its associated costs, availability and guarantees).

PlanObject: configuration (configuration parameters for the service tailored for the plan), availability (availability of the service for this plan expressed via time slots using the ISO 8601 time intervals format), pricing (specific pricing data for this plan. Overrides default pricing data defined before), quotas (specific quotas data for this plan. Overrides default quotas data defined before), rates (specific rates data for this plan. Overrides default

rates data defined before), guarantees (specific guarantees data for this plan. Overrides default guarantees data defined before).

QuotasObject: pathName (describes the API endpoint path quota configurations).

RatesObject: pathName (describes the API endpoint path rate configurations).

GuaranteesObject: pathName (describes a level of the guarantee supported by the plan).

GuaranteeObject: MethodName (an object describes the guarantee level).

GuaranteeObjectiveObject: objective (the objective of the guarantee), period (the period of the objective, that is, secondly, minutely, hourly, daily, monthly or yearly), window (the state of the Objective, that is, dynamic or static), scope (the scope of who request the service).

PathObject: methodName (the operations attached to this path).

OperationObject: metricName (the allowed limits of the request).

LimitObject: max (max value that can be accepted), period (the period of the objective, that is, secondly, minutely, hourly, daily, monthly or yearly), scope (the scope of who request the service).

ConfigurationsObject: name (configurations description).

4 Running example

In this section, we consider the example of a very simple RESTful service with a single endpoint (*/pets*) with two methods: *GET* and *POST* for the retrieving and creation of *pet* resources.

Codes 1.1 and 1.2 depict a comparison of the SLA modeling in iAgree and SLA4OAI. Specifically, the iAgree example defines both quotas and rates *over* requests by means of a `scope` for each tuple composed by a *plan* (e.g., free), *resource* (e.g., /pets), *operation* (e.g., post). Next, over this scope, some `limits` are being applied (e.g., a *maximum* of 10 in a daily *period*). Conversely, SLA4OAI is a bit more simpler, being not needed to specify a complex scope; defining, inside a specific `plan`, the `path` (e.g. /pets) and *method* (e.g., post) under the `rates` or `quotas` objects.

```
terms:
  pricing:
   ...
  configurations:
    ...
  metrics:
    requests:
      ...
      scope:
        resource: ...
        operation: ...

  quotas:
    - id: quotas_requests
      scope:
        plan: ...
        resource: ...
        operation: ...
      over:
        requests: ...
      of:
        - scope:
            plan: free
            resource: /pets
            operation: post
          limits:
            - max: 10
              period: daily
         ...

  rates:
    - id: rates_requests
      scope:
        plan: ...
        resource: ...
        operation: ...
      over:
        requests: ...
      of:
        - scope:
            plan: free
            resource: /pets
            operation: get
          limits:
            - max: 1
              period: secondly
```

Code 1.1. iAgree document example.

```
plans:
  free:
    pricing:
      cost: 10
    rates:
      /pets
        get:
          requests:
            - max: 1
              period: secondly
    quotas:
      /pets:
        post:
          requests:
            - max: 10
              period: daily
  pro:
    ...
```

Code 1.2. SLA4OAI document example.

5 Conclusions

In this paper, we have presented iAgree and SLA4OAI, two different proposals for the definition of pricing plans. On the one hand, iAgree is intended to model a large number of scenarios, including both human services, such as SLAs for business processes, and computational services, such as SLAs for RESTful APIs. Alternatively, SLA4OAI's goal is not model a wide range of SLA scenarios, but focus on a specific domain: RESTful APIs that specify limitations such as quotas and rates and are going to be modeled using a standard specification. In this context, we leverage of the extension points of the OpenAPI Specification (OAS) for building SLA4OAI that establishes a relationship between the API resources defined in the OAS document and the limitations (i.e., rates and quotas) for each one.

We plan to open the SLA4OAI model to the OpenAPI Initiative (OAI) community in order to obtain feedback and use it to improve the model and validate our proposal in other scenarios. This is one of the motivations of developing two SLA specifications, iAgree and SLA4OAI, in parallel. Furthermore, as part of the verification process, we plan to validate the iAgree model in different industrial scenarios derived from ICT projects.

As future work, we pretend to be able to model more sophisticated RESTful scenarios in which there exists a conversation between API consumers and providers.

References

1. Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Example-driven web API specification discovery. In *LNCS*, volume 10376 LNCS, pages 267–284. Springer, Cham, 7 2017.
2. Antonio Gamez-Diaz, Pablo Fernandez, and Antonio Ruiz-Cortes. An Analysis of RESTful APIs Offerings in the Industry. In *Service-Oriented Computing: 15th International Conference*, pages 589–604. 11 2017.
3. Antonio Gámez-Díaz, Pablo Fernández-Montes, and Antonio Ruiz-Cortés. Towards SLA-Driven API Gateways. In *Actas de las XI Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios*, 2015.
4. José María García, Octavio Martín-Díaz, Pablo Fernandez, Antonio Ruiz-Cortés, and Miguel Toro. Automated analysis of cloud offerings for optimal service provisioning. In *ICSOC 2017*. Springer, 2017.
5. Marc J Hadley. Web Application Description Language (WADL). *Search*, 12(TR-2006-153):1–31, 2006.
6. Octavio Martín-Díaz, Antonio Ruiz-Cortés, Amador Durán, and Carlos Müller. An Approach to Temporal-Aware Procurement of Web Services. In *ICSOC 2005*, volume LNCS 3826, pages 170–184. 2005.
7. T. Mauro. Microservices at Netflix: Lessons for Architectural Design.
8. Carlos Muller. *On the Automated Analysis of WS-Agreement Documents*. PhD thesis, Universidad de Sevilla, 2013.